NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN INVESTIGATION OF A FORTRAN GRAMMAR
FOR USE WITH A MICROPROCESSOR BASED
LALR(1) TRANSLATOR WRITING SYSTEM

by

Joan Marie Russell

March 1977

Thesis Advisor: Lyle V. Rich

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): An Investigation of a FORTRAN Grammar for use with a Microprocessor Based LALR(1) Translator Writing System	5. TYPE OF REPORT & PERIOD COVERED: Masters Thesis; March 1977 / 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Joan Marie Russell	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	12. REPORT DATE / 13. NUMBER OF PAGES: 70
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Naval Postgraduate School, Monterey, California 93940	15. SECURITY CLASS. (of this report): Unclassified / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Microcomputer Compiler
FORTRAN Grammar
FORTRAN Compiler
LALR(1) Grammar
Language Definition

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The design and implementation of an LALR(1) FORTRAN grammar has been described. Design requirements and recommendations for a 16K byte microcomputer system, which would allow the use of the FORTRAN programming language as defined by the grammar implementation, have been presented. The proposed system consisted of three subsystems: a FORTRAN compiler

DD FORM 1 JAN 73 1473 (Page 1) EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

based on the grammar implementation which produced an immediate language, a linking-loader that enabled independently compiled compiled program units to be linked, and an interpreter that executed the intermediate language on the specific target machine.

Approved for public release; distribution unlimited.

An Investigation of a FORTRAN Grammar for use with a Microprocessor based LALR(1) Translator Writing System

by

Joan Marie Russell
Lieutenant, United States Navy
B.A., Stetson University, 1971

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1977

ABSTRACT

The design and implementation of an LALR(1) FORTRAN grammar has been described. Design requirements and recommendations for a 16K byte microcomputer system, which would allow the use of the FORTRAN programming language as defined by the grammar implementation, have been presented. The proposed system consisted of three subsystems: a FORTRAN compiler based on the grammar implementation which produced an intermediate language, a linking-loader that enabled independently compiled program units to be linked, and an interpreter that executed the intermediate language on the specific target machine.

CONTENTS

I. INTRODUCTION

A. HISTORY OF THE FORTRAN LANGUAGE

FORTRAN was produced in the late 1950's for use on IBM computers. With the backing of IBM, FORTRAN became widely accepted and was subsequently developed for many machines during the 1960's. The American National Standards Committee specification of FORTRAN in 1966 [2] has gradually become accepted and most present compilers conform to this standard.

In 1976 the committee developed a draft proposed American National Standard FORTRAN [4] as a replacement for the original Standard. The FORTRAN language definition described in the proposed Standard included essentially all features of the original Standard with the major exception being the removal of the Hollerith data type. A number of additional capabilities including a character data type and file oriented input/output were also added to the language.

FORTRAN has made a significant contribution to computer technology. Its development provided a language that was easily learned by a wide variety of people and that was available for use on existing hardware. By providing a packed statement form which did not rely on the presence of blanks, FORTRAN allowed more efficient storage of programs

and greater ease of programming. With the use of the equivalence statement, the control of storage allocation by the programmer was permitted for the first time. [9]

Since its original definition, FORTRAN has become the standard scientific computer language. Because of the portability of programs written in FORTRAN it has also become a common intermediate language that has been generated by language processors and compilers, as well as one of the standard languages for program portability.

B. THE USE OF FORTRAN WITH MICROCOMPUTER SYSTEMS

Recent advances in the construction of digital circuits have resulted in the availability of low-cost LSI computer components. These components, which include central processing units, memory systems and peripherals for input/output, can be combined to form a digital computer known as a microcomputer. A large number of application areas for microcomputers have been identified, such as intelligent terminals, dedicated processors and minicomputer control tasks. [11]

In contrast to the advanced technology utilized in microcomputer hardware, the software designed to support microcomputers has been slow in developing. A great deal of applications work has been done directly in machine language since microcomputer configurations have often lacked the memory and input/output capacity to support program

development in assembly language. The use of assembly language has been supported by many microcomputers and when combined with a text editor and debugging aids formed a useful package for the programmer. To date, very few high-level languages have been developed for use on a microcomputer system. PLM [6] is currently the only high-level microcomputer systems programming language which is widely used.

With the expanding number of applications for microcomputers, high-level languages must assume an increasingly important role in the development of software for use on microcomputer systems. An implementation of the FORTRAN language could be a valuable addition to the high-level languages that can be utilized for microcomputer software support.

The purpose of this paper is to describe the design and implementation of an LALR(1) FORTRAN grammar for use with a self-hosted compiler. An overall system design to support the FORTRAN language on a microcomputer system is also described.

C. MOTIVATION FOR AN LALR(1) GRAMMAR

One of the major techniques used in current compiler construction is based on work done by Knuth [8], who developed deterministic parsing algorithms for the left-to-right translation of languages defined by LR(k) grammars. A

grammar is LR(k) if each sentence it generates can be parsed from left to right in a single scan with at most k lookahead symbols.

LR(k) grammars have several advantages. They are unambiguous. Construction algorithms exist for this class of grammar that can build parse action tables. A parser can use the tables produced by the analyzer to determine if language statements defined by the grammar are well-formed.

LR(k) grammars always require a lookahead of k symbols for the parser to determine the next state. LALR(k) grammars differ from LR(k) in that the lookahead is only performed when necessary, thus producing much smaller parse tables. The largest class of currently implementable LR(k) grammars are LALR(1).

An efficient parser can be written to interpret parse action tables for LALR(1) grammars [1]. The parser is a table-driven pushdown automaton that assumes a sequence of states (shift, reduce, accept, or error) while scanning the input. Decisions are based on the next input symbol and information accumulated on a parse stack. The final state indicates whether the input was well-formed.

The availability of such an LALR Parser Generator [10] for use in developing a FORTRAN grammar was the major factor in determining the method of constructing a compiler for use in the implementation of the FORTRAN language on a

microcomputer system. The LALR Parser program accepts a Backus Naur Form (BNF) grammar definition as input and the number of lookaheads allowed, and determines if the grammar is ambiguous. If the grammar is acceptable then parse tables are produced that can be used with a parser, and syntactic and semantic analyzer routines, to provide the basis for the systematic construction of a compiler. The parse tables that are produced are compatible with the PL/M programming language but can be modified for use with other languages.

The LALR Parser Generator was instrumental in the development of the large grammar necessary for FUPIRAN since BNF definitions could be tested and debugged incrementally as the grammar was developed.

II. GRAMMAR SPECIFICATION AND DESIGN

A. INTRODUCTION

This chapter describes the requirements, goals, and the design decisions considered during the development of the LALR(1) FORTRAN grammar. In addition, suggested extensions to the grammar are included.

The final two pass version of the LALR(1) FORTRAN grammar is contained in Appendices A and B. The syntax of the FORTRAN statements that this grammar defines is included in the 1978 draft proposed American National Standard FORTRAN. The few deviations from the proposed Standard are noted in the "Statement Restrictions" section in this chapter.

B. GRAMMAR SPECIFICATION

The syntax of individual FORTRAN statements and their correct ordering within program units described in the proposed Standard were used to form the basis of the grammar design. It should be noted that the grammar developed to define the proposed Standard also syntactically defines the 1966 ANSI Standard FORTRAN. Not considered in the design of the grammar were language extensions that have been made to

ANSI Standard FORTRAN by language processors, unless they have been included in the proposed Standard.

C. GRAMMAR DESIGN

FORTRAN as described in Refs. 2 and 4 has been considered an inherently ambiguous language. In order to completely define the syntax of the language an ambiguous grammar is required. Since LALR(1) grammars must be unambiguous by definition, this incompatibility created problems during the development of the grammar.

In the design of the grammar two approaches were taken in order to solve these problems. First, consideration was given to expanding the grammar to define more than the syntax allowed when compensating actions could be performed in the semantics of a compiler implementing the grammar. Second, if that approach failed then the grammar was restricted to define only a subset of the syntax of the language.

1. Design Goals

The design goals for the LALR(1) FORTRAN grammar were: (1) to adhere as closely as possible to the proposed ANS Standard requirements of the FORTRAN language definition, (2) to maintain overall simplicity in the grammar and (3) to develop a grammar small enough to be

utilized in a self-hosted compiler for a microcomputer system with 16K bytes of memory.

2. Tokens

The tokens in the initial grammar design consisted of special characters, reserved words, an identifier, a statement label, a format input, and character, integer, real and double precision constants. As the grammar was developed it was necessary to create statement termination, array identifier, exponentiation operator and concatenation operator tokens in order to resolve ambiguities.

a. Reserved Words

In order to recognize FORTRAN key words, such as DIMENSION, COMMON, READ, etc., the use of reserved words was required in the language definition. In the ANSI and proposed ANS Standard FORTRAN key words were not reserved and could also be used as identifiers. However, in order to conform to normal grammar techniques reserved word tokens were created to distinguish them from identifiers. In addition to the FORTRAN key words the logical constants .TRUE. and .FALSE., the relational operators .EQ., .NE., .GE., .GT., .LE. and .LT., and the logical operators .AND., .NOT., and .OR. were included as reserved words for ease in later implementation of the grammar.

b. Statement Termination

The FORTRAN language does not have a special "end-of-statement" delimiter equivalent to the period in COBOL or the semicolon in ALGOL. Thus, in order to terminate each statement definition in the grammar an "end-of-statement" token was created. Without this token, the LALR Parser Generator was unable to differentiate between individual statements in the language. The use of this token must be implemented in any compiler that utilizes the grammer but should be transparent to the user of the compiler.

c. Statement Labels

The special token "statement label" was used to define the statement labels given to specific statements. However, references to statement labels within a statement (e.g., GO TO 10) were defined as integer constants.

d. Special Characters

During the development of the grammar the initial set of special characters caused ambiguities in the definition of an expression. The differences in the use of the multiplication operator * and the exponentiation operator ** could not be resolved. A similar problem was encountered with the divide operator / and the concatenation operator //. It became necessary to create additional

tokens for the exponentiation operator and the concatenation operator.

e. Format Input

The "format input" token was included in the grammar design to allow format statements to be handled in the semantics of a compiler implementing the grammar, rather than in the grammar.

f. Read Paren

A major problem was encountered in developing the grammar to define the FORTRAN read statement. The syntax of the unformatted read statement was READ (<control information list>), while the syntax of the formatted read statement was READ <format>. With both the format and the control information list allowed to be an expression, a description of the syntax of the two read statements became READ (<expression>) and READ <expression>. Since the expression syntax included a rule that stated <expression> ::= (<expression>) there was no way for an LALR(1) grammar to unambiguously define both types of read statement. To solve this problem a "read paren" token was created to define the beginning of an unformatted read statement. Although it is syntactically correct to parenthesize the format in the formatted read, in utilizing the grammar the design imposes the requirement that a parenthesis following

the READ automatically indicates an unformatted read statement.

g. Identifiers and Array Identifiers

Identifiers were initially designed to be any sequence of one to six letters or numbers beginning with a letter, which was not a reserved word. However, a problem was encountered in differentiating between function references and array element references. The syntax of both as defined in the proposed Standard consists of an identifier followed by a parenthesized list of expressions, for example A(6,3,2) and MAX(6,3,2). Thus, in order to resolve this problem an array identifier token was created.

Distinguishing between identifiers and array identifiers remains a nontrivial problem and must be handled in the semantics. Depending on the technique used it may impose the requirement that arrays cannot be referenced prior to their definition in a dimension statement.

3. Expressions

The initial grammar design included the FORTRAN arithmetic, character and logical expressions as separate entities. These expressions are each constructed using identical operands - identifiers, array element references and function references. The specific type of each operand (character, integer, etc.) must be examined in order to determine whether it is valid for use in a particular

expression. The use of these identical operands again caused the grammar to be ambiguous. The solution was to define one general expression for overall use in the FORTRAN grammar. The rules that were developed for this expression definition enforce operator precedence for each type. The semantics of a compiler that uses such a grammar must be responsible for determining what specific type of expression is being used, and whether the operands are valid within that type of expression.

Another problem encountered in the expression definition was in enforcing parenthesized expressions as required in some FORTRAN statements. The syntax of an expression included the rule <expression> ::= (<expression>). This resulted in the reduction of a parenthesized expression to an expression prior to its use in a statement. In order to enforce a parenthesized expression the rule was modified as follows:

<expression> ::= <paren expression>

<paren expression> ::= (<expression>)

The second rule could then be used in any statements where parenthesized expressions were required.

4. Complex Constants

A further example that illustrates the problems encountered in constructing an LALR(1) grammar is the definition required for a complex constant. Syntactically a

complex constant was defined as (<real constant> , <real constant>). However, this definition could not be used in the grammar. Examination of the following grammar rules is necessary in order to understand the problem:

<complex constant> ::= (<real constant> ,
<real constant>)

<return statement> ::= RETURN <expression>

<expression> ::= <constant>
| <paren expression>

<paren expression> ::= (<expression>)

<constant> ::= <real constant>

Based on these grammar rules the beginning of one derivation for the return statement was RETURN (<real constant>. During the parsing of this statement with a left parenthesis and real constant on the stack the LALR Parser could not determine if the real constant should be reduced to a constant for eventual use in the return statement, or whether to stack a comma for eventual use in a complex constant.

In attempting to overcome the problem several alternative rules were examined for the complex constant definition that produced similar ambiguous results. The final unambiguous definition was as follows:

<complex constant> ::= <complex head> <expression>)

<complex head> ::= (<expression> ,

These rules require the semantics to determine if the

expressions in the complex constant definition are in fact real constants.

5. Input/Output Specification

The syntax of the FORTRAN input/output statements included a large number of input/output specifications associated with each statement, including unit numbers, error specifiers and file specifiers. The ordering of these specifers and the specific input/output specifications allowed with each statement were initially included in the grammar design. However, due to the large number of grammar rules required to enforce this syntax a general input/output specification replaced them in the final grammar. This requires the interpretation of specific input/output specifiers for the input/output statements in the semantics.

6. Statement Restrictions

The grammar for the individual FORTRAN statements was originally designed to strictly enforce the syntax of the statements in the proposed Standard. During the development of the grammar it was decided in several cases to define only a subset of the syntax in the grammar in order to decrease the number of rules. Both the common and data statement syntax enforced by the grammar allow only one namelist. Optional commas for the go to, type and go statements were also excluded from the grammar developed.

The implicit statement posed a special problem which was never entirely resolved. The length specification in the character implicit statement can be an expression and is defined by the following syntax:

<implicit statement> ::= IMPLICIT CHARACTER

* <expression> (<letter range list>)

The combination of an expression and a left parenthesis caused ambiguities in the grammar that could not be eliminated. The eventual solution was to restrict the syntax for the character length to an integer constant.

7. Optional Statement Design

If required for semantic analysis, many of the grammar rules in Appendices A and B that define the FORTRAN statements could be restructured and the overall FORTRAN grammar would still meet the requirements of an LALR(1) grammar. These alternate statement definitions might be useful in semantic code generation.

A simple example of this is illustrated by the following two alternate definitions available for the dimension statement:

<dimension stmt> ::= DIMENSION <array declaration>

| <dimension stmt> ,

<array declaration>

<dimension stmt> ::= <dimen head> <array declaration>

<dimen head> ::= DIMENSION

| <dimen head> <array declaration> ,

The first definition was chosen for use in the final grammar because it required fewer rules. The second set of rules may be desired for a compiler utilizing the grammar in order to determine when the last array declaration is being processed.

8. Splitting the Grammar

The original LALR(1) grammar was designed to define the syntax of all the statements in the FORTRAN language. The initial grammar definition that was developed contained approximately 350 rules. The tables generated by the LALR Parser for this grammar took over 11K bytes of memory. These tables were much too large to be implemented in a self-hosted compiler for a 16K microcomputer system with a 4K operating system. Consequently the grammar was split into two sections. The first section contained the rules for the data and environment definition statements including program, subroutine, function, block data, format, entry, data, specification and statement function statements. The second section contained the rules defining the format, entry, data and executable statements.

Splitting the grammar in this manner had two advantages. The large table size was reduced to 3800 bytes for section one and 4200 bytes for section two. The split grammar made it necessary to split the compiler into separate programs for each section; thus different semantic actions associated with identical grammar rules could be

varied within the separate programs. For example, a reference to an array element could be handled in a different manner in each of the programs.

A significant disadvantage of splitting the grammar was the difficulty imposed in the design of a compiler that utilizes the grammar to process more than one program unit.

The two grammars were designed so that they could easily be combined for use in a compiler that operated in an environment where memory size was not as restricted.

D. GRAMMAR AUGMENTATION

1. Overview

The initial grammar design included rules defining the relationships among program units, enforcing statement order and defining the statements allowed within the program units. These rules were subsequently dropped primarily to reduce the size of the parse tables. In an environment where the size of the compiler is not critical these rules would provide a useful extension to the grammar.

2. Program Units

The program units defined by the FORTRAN language are the main program, and the function, subroutine and block data subprograms. A FORTRAN program must have no more than one main program and can have any number of additional

subprograms. Further, these program units can be in any order. The LALR(1) grammar rules that were developed to enforce these relationships are as follows:

```
<program> ::= <program unit>
            | <subprogram>
            | <subprogram> <program unit>
<program unit> ::= <main program>
                 | <program unit> <subprogram unit>
<subprogram> ::= <subprogram unit>
               | <subprogram> <subprogram unit>
<suprogram unit> ::= <function subprogram>
                   | <subroutine subprogram>
                   | <block data subprogram>
```

These productions could be of value if more than one program unit is to be compiled at the same time.

3. Statement Ordering

Several versions of an LALR(1) grammar were developed to enforce statement ordering within program units and the types of statements permitted in each program unit. An LALR(1) grammar that met these requirements is presented in Appendix C. The parse tables generated for the grammar in Appendix C took approximately 2200 bytes of memory.

These rules could be included in a compiler that implements the grammar if the memory space required is available. An alternative would be to substitute the

appropriate semantic actions as is described in the design of the compiler presented later in this paper.

III. SYSTEM DESIGN

A. OVERVIEW

The system design recommended for implementation of the FORTRAN language on a microcomputer consists of three subsystems: a FORTRAN compiler that generates a relocatable intermediate language module for each program unit (main program, subroutines, functions, or block data) in the FORTRAN source file, a loader that links the modules that have been generated by the compiler, and an interpreter that executes the intermediate language.

The system that is described is designed to execute on the Intel 8080 microcomputer with 16K bytes of memory under the CP/M [3] operating system. CP/M is a monitor control program that provides a number of basic input/output functions, a console command processor, and a comprehensive file management package for use with a file system. The file system is maintained on diskettes (floppy disks) which contain 256K bytes of storage. This operating system also supports a text editor, a dynamic debugger and the Intel 8080 assembler. CP/M takes 4K bytes of memory; therefore the system design discussed for the implementation of FORTRAN has 12K bytes of memory available. The use of CP/M or an equivalent system on the 8080 microcomputer directly

affected the design requirements and recommendations made for the implementation of FORTRAN.

B. COMPILER

1. Organization

Splitting the grammar into two sections, as noted previously, had a direct impact on the compiler design. The compiler was originally envisioned as one program with provisions for multiple passes. The implementation of the split FORTRAN grammar required a separate program for each grammar and a control program that provided linkage between the two programs.

To execute the compiler the user of the system would invoke the control program, and pass the name of the source file to be compiled in the command line as a parameter to the program. The control program would then manage the interfaces necessary between the pass 1 and pass 2 compiler programs required in the compilation process. The final output would be a file containing the intermediate language generated by the compiler.

The system is designed so that the control program resides in memory during the entire compilation. The symbol table area is left in memory for use by the pass 2 program after the pass 1 program has completed execution. The pass 2 program overlays the pass 1 program when read into memory

by the control program. The memory configuration for the implementation of this design is presented in Figure 1.

COMPILER MEMORY ORGANIZATION

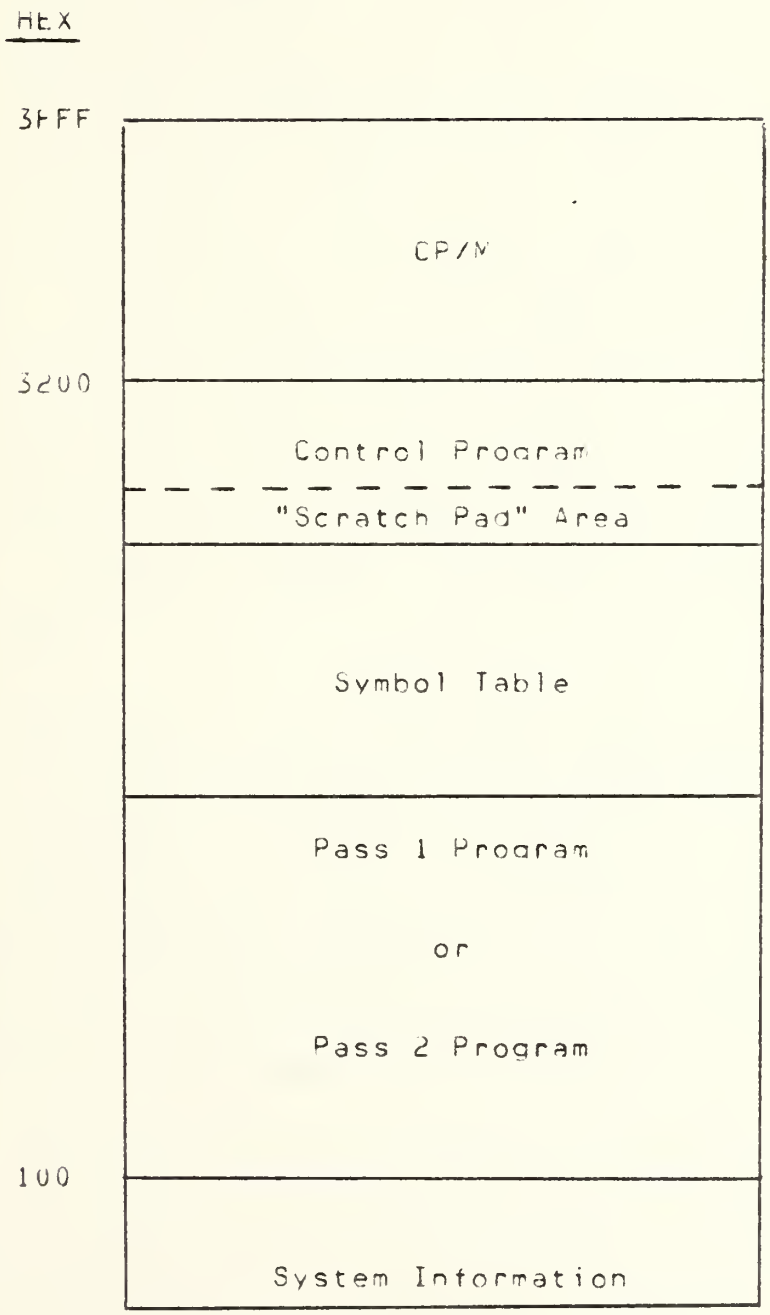

Figure 1

This section discusses the functions, the design requirements and recommendations for the control program, the pass 1 and pass 2 programs, and the interfaces required among them necessary to implement the FORTRAN compiler based on the LALR(1) FORTRAN grammar.

2. Control Program

The main purpose of the control program is to control the overall compilation process. In order to accomplish this it must perform two basic functions: (1) the loading of the pass 1 and pass 2 programs and the initialization of their execution, and (2) the maintenance of common information such as compiler togales and symbol table necessary to both passes. This requires that the control program remain in memory during the entire compilation.

The bulk of the executable code for the control program resides in memory just below the CP/M operating system (see Figure 1). Upon initiation of the program by the user, execution begins at 100 hex (100H) and an immediate jump is performed to the first executable byte of code located in the upper part of memory.

The first task the control program must perform is to decide whether it is being invoked from either the pass 1 or pass 2 program or at the initiation of the FORTRAN compiler. The appropriate actions can then be provided

based on this decision. A control flag which can be altered by the pass 1 and pass 2 programs can be used to implement this requirement.

When the control program is executed for the initialization of a compilation it should perform the following functions: (1) initialize its "scratch pad" area for use by the pass 1 and pass 2 programs, (2) save the file control block for the FORTRAN source file and open the file for input, (3) maintain the file control block for the intermediate language file and open it for output, (4) initialize the symbol table area, (5) read the executable (COM) file for the pass 1 program into memory beginning at 100H, and (6) jump to 100H to transfer control to the pass 1 program.

When the control program is invoked at the completion of the pass 1 program it should check for a fatal error in the pass 1 phase of compilation which would terminate execution. If none is found, the COM file for the pass 2 program can be read into memory and control transfered to 100H to begin execution of the pass 2 program.

When the control program is executed via a transfer from the pass 2 program it should again check for a fatal error in the pass 2 phase of compilation and terminate execution if necessary. It must also determine if another program unit is to be compiled. If an additional program unit is to be compiled the control program must reinitialize

the symbol table and "scratch pad" area, with the exception of compiler toggles, and reload and transfer control back to the pass 1 program to continue the compilation process. If no more program units are present on the source file, the control program must close the FORTRAN source file and the intermediate language file and return control to the CP/M operating system.

Maintenance of the file control blocks by the control program for both the FORTRAN source file and intermediate language file is critical to the system. Pointers must be maintained for both files in order to determine the correct record to be processed for input or output.

The "scratch pad" area located in the control program is available for use by both the pass 1 and pass 2 programs. Information maintained in this area can include compiler toggles, error flags, and any other interface information required by the pass 1 and pass 2 programs.

3. Pass 1 Program

The pass 1 program implements the grammar presented in Appendix A. This program processes the FORTRAN statements up to (but not including) the first executable statement. Routines for syntactic and semantic analysis, symbol table manipulation, and a parser must be included in the program. This section discusses the design requirements

imposed by the FORTRAN grammar and some additional design considerations necessary for implementation of the program. The parser and scanner described in this section were implemented and tested.

a. Parser

The parser that was adopted for use with the pass 1 program is based on the parse action generation algorithms used to analyze LALR(1) grammars.

The parser controls the execution of the pass 1 program. It receives a series of tokens from the scanner and analyzes them to determine if they form a valid sentence in the FORTRAN grammar. It bases this decision on the next input token and information previously accumulated on a parse stack where the parser states are maintained.

The basic actions performed by the parser include a shift action that reads a new token and pushes the previous state onto the stack, a reduce state that pops the number of elements equal to the handle of the production and puts a new state on the stack, an accept state that indicates the input conforms to the grammar, and an error state that indicates when a syntax error has occured. Additional stacks can be used in parallel with the parse stack that relate to the translation of the program, such as pointers into the symbol table and temporary values used in reductions.

In order to stop the execution of the pass 1 program the grammar allows the parser to proceed until it has analyzed an end statement, when no executable statements are found in the program unit, until it has analyzed the reserved word in the first executable statement, such as DO or READ, or until an assignment statement is recognized. In the case of the assignment statement, where no reserved word is contained in the statement (e.g., A = 3), it parses up to the equal sign. The information previously scanned for the executable statement must be saved for use by the pass 2 program to provide initialization of the scanner and to allow for any semantic actions that need to be performed. By maintaining a stack that always contains the last three tokens processed, this information can then be provided to the pass 2 program via the "scratch pad" in the control program.

b. Scanner

The function of the scanner is to provide the tokens defined in the FORTRAN grammar to the parser. These tokens include reserved words, special characters, the exponentiation and concatenation operators, statement labels, identifiers, array identifiers, "format inputs", "end-of-statements", and integer, real, character, and double precision constants. The scanner that was implemented in the pass 1 program encountered no special problems in recognizing these tokens with the exception of

identifiers, array identifiers and the "end-of-statement" token.

Identifiers and array identifiers both have the same structure. They are a sequence of one to six letters or digits that begin with a letter. In order to differentiate between them, it was necessary to include interaction with the semantics necessary for processing dimension statements. When the reserved word DIMENSION was encountered a flag was set to indicate that a token of this form followed by a left parentnesis was an array identifier. This flag was checked by the scanner following the initial test for reserved words. If the flag was not set, the symbol was looked up in the symbol table to determine if it was an array identifier defined in a previous dimension statement. If these tests failed, the token was assumed to be an identifer. The use of this technique imposed the requirement that arrays could not be referenced in any FORTRAN statement prior to their declaration in a dimension statement.

The recognition of the end of a FORTRAN statement by the LALR(1) grammar implementation required an "end-of-statement" token transparent to the user. A lookahead feature was used in the scanner to help determine whether the next line was a continuation of the previous statement. Since normal FORTRAN card conventions were maintained, the decision could be based on a line position

pointer that maintained the "card column" position of the current symbol being considered by the scanner. When a carriage return and linefeed were encountered, the position of the next nonblank character was determined. If the position was not "card column" six, an "end-of-statement" token was passed to the parser. The line position pointer was also used to recognize the end of valid statement input at "card column" seventy-two.

c. Semantic Analysis

As noted previously, the grammar for the pass 1 program does not enforce the order and the types of statements allowed in each program unit. Order can be enforced in the semantics of the program by the use of two flags: one flag to determine the type of program unit being processed (main program, subroutine, function, or block data), and a second flag to determine if a particular statement is valid based on the previous statements that have been processed. Each statement in the grammar for this program has an associated reserved word. Whenever a reserved word for the statement currently being processed is encountered, the flags can be checked to determine if the statement order is correct and if that type of statement is valid in the program unit.

The use of the "format input" token in the grammar requires the processing of format statements in the semantics of the program. In addition, this information

must be saved for later use by the pass 2 program in the processing of the executable statements. This can be accomplished by either writing the information required to a floppy disk file, or by saving the information in the "scratch pad" area of the control program. Since the number of format statements may be large, the exact implementation must be based on the actual memory available for use in the control program.

The general expression definition in the FORTRAN grammar has a direct impact on the pass 1 program. The semantics must enforce the type of expression (character, logical, or arithmentic) allowed within each statement of the FORTRAN input. In some cases, such as dimension statements, only integer constant expressions are valid. Integer constant expressions are a special case of the arithmetic expression in which only integer constants or variables of type integer are allowed. The semantics of the compiler must also process these special expressions and provide for their evaluation and use.

d. Code Generation

The type of code generation produced by a compiler is highly dependent on the system in which it is implemented. The design decision to produce an intermediate language instead of executable machine code was based on two major considerations. First, the production of an intermediate language enhances the transportability of an

eventual system implementation of FORTRAN to other microcomputers that support PL/M. Second, the existence of an interpreter of Basic-E [5], which translates an intermediate language output from the Basic-E compiler, has already been successfully implemented in the computer laboratory at the Naval Postgraduate School. This interpreter is an excellent candidate for modification for use with FORTRAN if the intermediate language produced by the FORTRAN compiler is compatible with the Basic-E intermediate language.

4. Pass 2 Program

The pass 2 program implements the grammar presented in Appendix B. This program processes FORTRAN executable statements. Syntactic and semantic analysis, symbol table manipulation, and parser routines must again be included in the program. This section describes the design requirements imposed by the FORTRAN grammar and additional design considerations necessary for implementation of the program.

a. Parser

The parser, which controls the execution of the pass 2 program, can be identical to the parser described in the previous section.

Execution of the parser is terminated after the end statement is parsed. At this point the program must determine if there are additional program units to be

compiled. This can be done by checking to see if anything other than a soft end-of-file (IAM) or a hard end-of-file occurs after the carriage return and linefeed following the end statement. The appropriate flag should then be set in the "scratch pad" area of the control program and execution transferred to the control program.

b. Scanner

The scanner designed for use in the pass 2 program can be very similar to the scanner in the pass 1 program. The "read paren" token is the only additional token that must be recognized by the pass 2 scanner implementation.

The differentiation between identifiers and array identifiers is no longer required in the semantic analysis. At this point all arrays have been declared and the array identifiers are contained in the symbol table and can be easily recognized.

At the initialization of the scanner, the tokens previously parsed in the pass 1 program for the first executable statement must be recovered from the "scratch pad". Code for providing these tokens for analysis and use by the scanner must be included in the program prior to obtaining any new tokens for the first executable statement.

c. Semantic Analysis

As noted previously, the format specification in the format statement must be handled in the semantics of the program. In addition, provision must be made for retrieving the information that was produced for any format statements that were processed by the pass 1 program.

The grammar for pass 2 imposes additional requirements for expression evaluation not necessary in the pass 1 program. For example, one form of the print statement which is acceptable to the grammar is PRINT <expression>. The expression may either be an integer constant designating a statement label, or a character expression. Thus, the semantics must allow the statement label to be valid as it is parsed up through the expression definition associated with a print statement. Similar requirements exist for the read statement and complex constant definitions.

d. Code Generation

Since the pass 2 program performs code generation for all FORTRAN executable statements, the program may exceed the memory size available. If this occurs, consideration should be given to either restricting the types of statements allowed for use in the FORTRAN implementation or to producing parse actions in pass 2 and adding a pass 3 program to process these parse actions. The

additional program could then generate the intermediate language based on the parse actions, associated information and available symbol table information.

C. LOADER

The basic task of the loader program is to process the intermediate language modules generated by the compiler for the various program units, and to produce a zero-address intermediate language module that can be executed by the interpreter.

The following types of information associated with each intermediate language module are necessary for loader implementation: (1) the name of the current module, (2) a list of external names and references with definitions of their use, (3) the address of the first byte in the code area of the current module, and (4) the length of the code area of the module in bytes. Output from the loader should be designed to enable further linkage if all external references have not yet been resolved.

The actual implementation of a loader was not considered part of this thesis project and is left for future consideration.

D. INTERPRETER

The function of the interpreter is to execute the zero-address intermediate language produced by either the compiler or the loader. At this point all external references must be resolved in order for the intermediate language module to be interpreted.

The design of an interpreter is dependent on the specific machine on which the FORTRAN language is to be implemented. The run-time monitor used for executing the intermediate language produced by the Basic-E compiler [5] is an example of an interpreter that has been successfully implemented on the 8080 under the CP/M operating system. The monitor provides a number of features that would be useful in the interpretation of FORTRAN such as the use of a floating point package [7] to perform arithmetic, function evaluation and conversion operations on 32 bit floating point numbers. If the intermediate language generated by the FORTRAN compiler is designed to be compatible with the language produced by the Basic-E compiler, the modification of this interpreter to accept FORTRAN would greatly facilitate the implementation of FORTRAN on the 8080.

IV. CONCLUSIONS

The successful completion of the formal FORTRAN grammar demonstrates the feasibility of defining an ambiguous language, such as FORTRAN, using an LALR(1) grammar. Ambiguities in the grammar can be resolved by providing a broader definition for the language and compensating semantic actions by a compiler that implements the grammar.

The FORTRAN grammar which was developed was structured to define the largest possible syntax of the 1978 draft proposed American National Standard FORTRAN. However, this should not prevent a user of this grammar from redefining it to meet the requirements for implementation on a particular machine.

The use of a formal language and automatic parser generation methods proved extremely valuable in the construction of the FORTRAN compiler. The parser that was available for use in processing the parse tables, when combined with syntactic and semantic analyzer routines, led to a modular design and the systematic construction of a compiler rather than an ad hoc technique.

The system design which was presented to support the FORTRAN language on a microcomputer system with 16K bytes of memory is feasible. However, the lack of memory space

remains a problem. It is recommended that consideration be given to the replacement of those rules, especially input/output statements, which could be tailored to the specific machine on which the compiler is implemented.

It is hoped that the LALR(1) FORTRAN grammar and the accompanying system design recommendations will establish a basis for the implementation of FORTRAN on a microcomputer system.

APPENDIX A - FORTRAN GRAMMAR SECTION ONE

```
<program> ::= <prog stmt> <program body> <end state>
            | <prog stmt> <end state>
            | <program body> <end state>
            | <end state>
            | <subr stmt> <program body> <end state>
            | <subr stmt> <end state>
            | <func stmt> <program body> <end state>
            | <func stmt> <end state>
            | <block data stmt> <program body> <end state>
<program body> ::= <statement>
                 | <program body> <statement>
<statement> ::= <label> <parm stmt> <eos>
              | <label> <impl stmt> <eos>
              | <label> <dimen stmt> <eos>
              | <label> <common stmt> <eos>
              | <label> <equiv stmt> <eos>
              | <label> <type stmt> <eos>
              | <label> <external stmt> <eos>
              | <label> <intrinsic stmt> <eos>
              | <label> <save stmt> <eos>
              | <label> <data stmt> <eos>
              | <label> <stmtfunc stmt> <eos>
```

```
                | <label> <entry stmt> <eos>
                | <stmt label> <format stmt> <eos>
<end state> ::= <label> <exec stmt reserved word>
                | <label> <identifier> =
                | <label> <array element> =
                | <label> <substring name> =
                | <end stmt>
<exec stmt reserved word> ::= DO
                            | IF
                            | ASSIGN
                            | GO
                            | CONTINUE
                            | STOP
                            | PAUSE
                            | CALL
                            | READ
                            | WRITE
                            | PRINT
                            | OPEN
                            | CLOSE
                            | INQUIRE
                            | BACKSPACE
                            | ENDFILE
                            | REWIND
                            | RETURN
<end stmt> ::= END
<prog stmt> ::= <label> PROGRAM <identifier> <eos>
```

```
<block data stmt> ::= <label> BLOCK DATA <eos>
                    | <label> BLOCK DATA <identifier> <eos>
<subr stmt> ::= <label> SUBROUTINE <identifier> <eos>
              | <label> SUBROUTINE <arg list> <eos>
<func stmt> ::= <label> <func id>
              | <label> <number type> <func id>
              | <label> <char type> <func id>
<func id> ::= FUNCTION <identifier> <eos>
            | FUNCTION <identifier> ( ) <eos>
            | FUNCTION <arg list> <eos>
<parm stmt> ::= PARAMETER <identifier> = <constant>
              | <parm stmt> , <identifier> = <constant>
<impl stmt> ::= IMPLICIT <impl list>
              | <impl stmt> , <impl list>
<impl list> ::= <impl list head> <letter range> )
<impl list head> ::= <number type> (
                   | CHARACTER (
                   | CHARACTER * <integer constant> (
                   | <impl list head> <letter range> ,
<letter range> ::= <identifier>
                 | <identifier> - <identifier>
<dimen stmt> ::= DIMENSION <array decl>
               | <dimen stmt> , <array decl>
<common stmt> ::= COMMON <common name> <common nlist item>
                | <common stmt> , <common nlist item>
```

```
<common name> ::= <empty>
                | <label common name>
                | <double slash>
<common nlist item> ::= <identifier>
                      | <array id>
                      | <array decl>
<label common name> ::= / <identifier> /
<equiv stmt> ::= EQUIVALENCE <equiv nlist>
               | <equiv stmt> , <equiv nlist>
<equiv nlist> ::= <equiv nlist head> <equiv nlist item> )
<equiv nlist head> ::= ( <equiv nlist item> ,
                     | <equiv nlist head>
                          <equiv nlist item> ,
<equiv nlist item> ::= <identifier>
                     | <array id>
                     | <array element>
                     | <substring name>
<type stmt> ::= <number type stmt>
              | <char type stmt>
<number type stmt> ::= <number type> <type item>
                     | <number type stmt> , <type item>
<type item> ::= <identifier>
              | <array id>
              | <array decl>
<char type stmt> ::= <char type> <char name>
                   | <char type stmt> , <char name>
```

```
<char name> ::= <identifier>
              | <identifier> * <char len>
              | <array decl>
              | <array decl> * <char len>
              | <array id>
              | <array id> * <char len>
<external stmt> ::= EXTERNAL <identifier>
                  | <external stmt> , <identifier>
<intrinsic stmt> ::= INTRINSIC <identifier>
                   | <intrinsic stmt> , <identifier>
<save stmt> ::= SAVE
              | <save list>
<save list> ::= SAVE <save item>
              | <save list> , <save item>
<save item> ::= <identifier>
              | <array id>
              | <label common name>
<data stmt> ::= <data list> <data clist item> /
<data list> ::= <data head> <data nlist item> /
              | <data list> <data clist item> ,
<data head> ::= DATA
              | <data head> <data nlist item> ,
<data nlist item> ::= <identifier>
                    | <array id>
                    | <array element>
                    | <substring name>
                    | <implied do list>
```

```
<data clist item> ::= <identifier>
                    | <constant>
                    | <integer constant> * <constant>
                    | <integer constant> * <identifier>
                    | <identifier> * <constant>
                    | <identifier> * <identifier>
<implied do list> ::= ( <array element> , <do list> )
                    | ( <implied do list> , <do list> )
<stmtfunc stmt> ::= <arg list> = <exp>
                  | <identifier> ( ) = <exp>
<entry stmt> ::= ENTRY <identifier>
               | ENTRY <arg list>
               | ENTRY <identifier> ( )
<format stmt> ::= FORMAT <format input>
<do list> ::= <identifier> = <exp> , <exp>
            | <identifier> = <exp> , <exp> , <exp>
<func ref> ::= <identifier> ( )
             | <arg list>
<arg list> ::= <arg head> )
<arg head> ::= <identifier> ( <arg element>
             | <arg head> , <arg element>
<arg element> ::= <exp>
                | <array id>
                | *
<array decl> ::= <array id> <dimen decl list> <dimen decl )
<dimen decl list> ::= (
                    | <dimen decl list> <dimen decl> ,
```

```
<dimen decl> ::= <exp>
               | <exp> : <exp>
<array element> ::= <array element list> <exp> )
<aray element list> ::= <array id> (
                      | <array element list> <exp> ,
<substring name> ::= <identifier> ( <substring decl>
                   | <array element> ( <substring decl>
<substring decl> ::= <exp> : <exp> )
                   | <exp> : )
                   | : <exp> )
                   | : )
<exp> ::= <logical term>
        | <exp> .OR. <logical term>
<logical term> ::= <logical factor>
                 | <logical term> .AND. <logical factor>
<logical factor> ::= <logical primary>
                   | .NOT. <logical primary>
<logical primary> ::= <char exp>
                    | <char exp> <rel op> <char exp>
<char exp> ::= <arith exp>
             | <char exp> <double slash> <arith exp>
<arith exp> ::= <arith term>
              | + <arith term>
              | - <arith term>
              | <arith exp> + <arith term>
              | <arith exp> - <arith term>
```

```
<arith term> ::= <arith factor>
               | <arith term> / <arith factor>
               | <arith term> * <arith factor>
<arith factor> ::= <arith primary>
                 | <arith factor> <expon op> <arith primary>
<arith primary> ::= <constant>
                  | <identifier>
                  | <array element>
                  | <substring name>
                  | <func ref>
                  | <paren exp>
<paren exp> ::= ( <exp> )
<constant> ::= <integer constant>
             | <real constant>
             | <dble pre constant>
             | <logical constant>
             | <char constant>
             | <complex constant>
<complex constant> ::= ( <real constant> , <real constant> )
<rel op> ::= .LT.
           | .LE.
           | .EQ.
           | .NE.
           | .GT.
           | .GE.
<logical constant> ::= .TRUE.
                     | .FALSE.
```

```
<number type> ::= INTEGER
                | REAL
                | DOUBLE PRECISION
                | COMPLEX
                | LOGICAL
<char type> ::= CHARACTER
              | CHARACTER * <char len>
<char len> ::= <paren exp>
             | <integer constant>
             | ( * )
<label> ::= <empty>
          | <stmt label>
```

APPENDIX B - FORTRAN GRAMMAR SECTION TWO

```
<program> ::= <program body> <end stmt>

<program body> ::= <statement>
                 | <program body> <statement>

<statement> ::= <label> <data stmt> <eos>
              | <label> <logif stmt>
              | <label> <do stmt> <eos>
              | <label> <entry stmt> <eos>
              | <stmt label> <format stmt> <eos>
              | <logif exec stmt>

<logif exec stmt> ::= <label> <assign stmt> <eos>
                    | <label> <goto stmt> <eos>
                    | <label> <aritnif stmt> <eos>
                    | <label> <continue stmt> <eos>
                    | <label> <stop stmt> <eos>
                    | <label> <pause stmt> <eos>
                    | <label> <call stmt> <eos>
                    | <label> <return stmt> <eos>
                    | <label> <read write print stmt> <eos>
                    | <label> <open close inquire stmt> <eos>
                    | <label> <backspace endfile rewind stmt>
                          <eos>

<end stmt> ::= END

<data stmt> ::= <data list> <data clist item> /
```

```
<data list> ::= <data head> <data nlist item> /
              | <data list> <data clist item> ,
<data head> ::= DATA
              | <data head> <data nlist item> ,
<data nlist item> ::= <identifier>
                    | <array id>
                    | <array element>
                    | <substring name>
                    | <implied do list>
<data clist item> ::= <identifier>
                    | <constant>
                    | <integer constant> * <constant>
                    | <integer constant> * <identifier>
                    | <identifier> * <constant>
                    | <identifier> * <identifier>
<implied do list> ::= ( <array element> , <do list> )
                    | ( <implied do list> , <do list> )
<pause stmt> ::= PAUSE
               | PAUSE <integer constant>
               | PAUSE <char constant>
<stop stmt> ::= STOP
              | STOP <integer constant>
              | STOP <char constant>
<continue stmt> ::= CONTINUE
<return stmt> ::= RETURN
                | RETURN <exp>
<arithif stmt> ::= IF <paren exp> <aif slabels>
```

```
<aif slabels> ::= <integer constant> , <integer constant> ,
                  <integer constant>
<logif stmt> ::= IF <paren exp> <logif exec stmt>
<assign stmt> ::= ASSIGN <integer constant> TO <identifier>
                | <identifier> = <exp>
                | <array element> = <exp>
                | <substring name> = <exp>
<do stmt> ::= DO <integer constant> <do list>
<goto stmt> ::= GO TO <integer constant>
              | GO TO <stmt label list> ) <exp>
              | GO TO <identifier>
              | GO TO <identifier> <stmt label list> )
<stmt label list> ::= ( <integer constant>
                      | <stmt label list> , <integer constant>
<call stmt> ::= CALL <identifier>
              | CALL <arg list>
<entry stmt> ::= ENTRY <identifier>
               | ENTRY <arg list>
               | ENTRY <identifier> ( )
<format stmt> ::= FORMAT <format input>
<open close inquire stmt> ::= <open close inquire head>
                                  <exp> )
                            | <open close inquire head>
                                  <io spec> )
<open close inquire head> ::= OPEN (
                            | CLOSE (
                            | INQUIRE (
```

```
                                | <open close inquire head>
                                      <exp> ,
                                | <open close inquire head>
                                      <io spec> ,
<backspace endfile rewind stmt> ::= BACKSPACE <ber list>
                                  | ENDFILE <ber list>
                                  | REWIND <ber list>
<ber list> ::= <exp>
             | ( <exp> , <io spec> )
             | ( <io spec> , <exp> )
<read write print stmt> ::= <read print stmt>
                          | <read write ci list>
                          | <read write io list>
<read print stmt> ::= READ <exp>
                    | READ <array id>
                    | READ *
                    | PRINT <exp>
                    | PRINT <array id>
                    | PRINT *
                    | <read print stmt> , <io list item>
<read write io list> ::= <read write ci list> <io list item>
                       | <read write io list> ,
                            <io list item>
<read write ci list> ::= <read write head> <ci list item> )
<read write head> ::= <read paren>
                    | WRITE (
                    | <read write head> <ci list item> ,
```

```
<cl list item> ::= <exp>
                 | <io spec>
                 | <array id>
                 | *
<array block item> ::= <array element> : <array element>
                     | <array element> :
                     | : <array element>
<io implied do list> ::= <complex head> <do list> )
                       | <io do list head> <do list> )
<io do list head> ::= <complex head> <exp> ,
                    | ( <array id> ,
                    | ( <array block item> ,
                    | ( <io implied do list> ,
                    | <io do list head> <io list item> ,
<io list item> ::= <exp>
                 | <array id>
                 | <array block item>
                 | <io implied do list>
<io spec> ::= UNIT = <exp>
            | ERR = <integer constant>
            | REC = <exp>
            | END = <integer constant>
            | FMT = <array id>
            | FMT = <exp>
            | FMT = *
            | FILE = <exp>
            | STATUS = <exp>
```

```
            | BLANK = <exp>
            | ACCESS = <exp>
            | FORM = <exp>
            | RECL = <exp>
            | MAXREC = <exp>
            | EXIST = <exp>
            | OPENED = <exp>
            | NUMBER = <exp>
            | NAMED = <exp>
            | NAME = <exp>
            | NEXTREC = <exp>
<do list> ::= <identifier> = <exp> , <exp>
            | <identifier> = <exp> , <exp> , <exp>
<func ref> ::= <identifier> ( )
             | <arg list>
<arg list> ::= <arg head> )
<arg head> ::= <identifier> ( <arg element>
             | <arg head> , <arg element>
<arg element> ::= <exp>
                | <array id>
                | * <integer constant>
                | *
<array element> ::= <array element list> <exp> )
<aray element list> ::= <array id> (
                      | <array element list> <exp> ,
<substring name> ::= <identifier> ( <substring decl>
                   | <array element> ( <substring decl>
```

```
<substring decl> ::= <exp> : <exp> )
                   | <exp> : )
                   | : <exp> )
                   | : )
<exp> ::= <logical term>
        | <exp> .OR. <logical term>
<logical term> ::= <logical factor>
                 | <logical term> .AND. <logical factor>
<logical factor> ::= <logical primary>
                   | .NOT. <logical primary>
<logical primary> ::= <char exp>
                    | <char exp> <rel op> <char exp>
<char exp> ::= <arith exp>
             | <char exp> <double slash> <arith exp>
<arith exp> ::= <arith term>
              | + <arith term>
              | - <arith term>
              | <arith exp> + <arith term>
              | <arith exp> - <arith term>
<arith term> ::= <arith factor>
               | <arith term> / <arith factor>
               | <arith term> * <arith factor>
<arith factor> ::= <arith primary>
                 | <arith factor> <expon op> <arith primary>
<arith primary> ::= <constant>
                  | <identifier>
                  | <array element>
```

```
                    | <substring name>
                    | <func ref>
                    | <paren exp>
<paren exp> ::= ( <exp> )
<constant> ::= <integer constant>
             | <real constant>
             | <dble pre constant>
             | <logical constant>
             | <char constant>
             | <complex constant>
<complex constant> ::= <complex head> <exp> )
<complex head> ::= ( <exp> ,
<rel op> ::= .LT.
           | .LE.
           | .EQ.
           | .NE.
           | .GT.
           | .GE.
<logical constant> ::= .TRUE.
                     | .FALSE.
<label> ::= <empty>
          | <stmt label>
```

APPENDIX C - FORTRAN GRAMMAR FOR STATEMENT ORDER

```
<program> ::= <program unit>
            | <subprogram>
            | <subprogram> <program unit>
<program unit> ::= <main program>
                 | <program unit> <subprogram unit>
<subprogram> ::= <subprogram unit>
               | <subprogram> <subprogram unit>
<subprogram unit> ::= <function subprogram>
                    | <subroutine subprogram>
                    | <block data subprogram>
<main program> ::= <prog stmt> <main program body>
                 | <main program body>
<subroutine subprogram> ::= <subr stmt> <sub program body>
                          | <subr stmt> <main program body>
                          | <subr stmt> <block data body>
                          | <subr stmt> <end stmt>
<function subprogram> ::= <func stmt> <sub program body>
                        | <func stmt> <main program body>
                        | <func stmt> <block data body>
                        | <func stmt> <end stmt>
<block data subprogram> ::= <block data stmt>
                                <block data body>
                          | <block data stmt> <end stmt>
```

```
<main program body> ::= <main4 exec> <end stmt>

<subprogram body> ::= <main1 impl> <end stmt>
                    | <main2 spec> <end stmt>
                    | <main3 func> <end stmt>
                    | <sub1 impl> <end stmt>
                    | <sub2 spec> <end stmt>
                    | <sub3 func> <end stmt>
                    | <sub4 exec> <end stmt>
                    | <sub5 return> <end stmt>

<block data body> ::= <blok1 impl> <end stmt>
                    | <blok2 spec> <end stmt>
                    | <blok3 data> <end stmt>

<blok1 impl> ::= <impl stmt>
               | <parm stmt>
               | <blok1 impl> <impl stmt>
               | <blok1 impl> <parm stmt>

<blok2 spec> ::= <spec stmt>
               | <blok1 impl> <spec stmt>
               | <blok2 spec> <spec stmt>
               | <blok2 spec> <parm stmt>

<blok3 data> ::= <data stmt>
               | <blok1 impl> <data stmt>
               | <blok2 spec> <data stmt>
               | <blok3 data> <data stmt>

<main1 impl> ::= <format stmt>
               | <blok1 impl> <format stmt>
               | <main1 impl> <impl stmt>
```

| <main1 impl> <parm stmt>

| <main1 impl> <format stmt>

<main2 spec> ::= <other spec stmt>

| <blok1 impl> <other spec stmt>

| <blok2 spec> <other spec stmt>

| <blok2 spec> <format stmt>

| <main1 impl> <other spec stmt>

| <main1 impl> <spec stmt>

| <main2 spec> <other spec stmt>

| <main2 spec> <spec stmt>

| <main2 spec> <parm stmt>

| <main2 spec> <format stmt>

<main3 func> ::= <stmtfunc stmt>

| <blok1 impl> <stmtfunc stmt>

| <blok2 spec> <stmtfunc stmt>

| <blok3 data> <stmtfunc stmt>

| <blok3 data> <format stmt>

| <main1 impl> <stmtfunc stmt>

| <main1 impl> <data stmt>

| <main2 spec> <stmtfunc stmt>

| <main2 spec> <data stmt>

| <main3 func> <stmtfunc stmt>

| <main3 func> <data stmt>

| <main3 func> <format stmt>

<main4 exec> ::= <exec stmt>

| <blok1 impl> <exec stmt>

| <blok2 spec> <exec stmt>

| <blok3 data> <exec stmt>

| <main1 impl> <exec stmt>

| <main2 spec> <exec stmt>

| <main3 func> <exec stmt>

| <main4 exec> <exec stmt>

| <main4 exec> <data stmt>

| <main4 exec> <format stmt>

<sub1 impl> ::= <entry stmt>

| <blok1 impl> <entry stmt>

| <main1 impl> <entry stmt>

| <sub1 impl> <impl stmt>

| <sub1 impl> <parm stmt>

| <sub1 impl> <format stmt>

| <sub1 impl> <entry stmt>

<sub2 spec> ::= <save stmt>

| <blok1 impl> <save stmt>

| <blok2 spec> <save stmt>

| <blok2 spec> <entry stmt>

| <main1 impl> <save stmt>

| <main2 spec> <save stmt>

| <main2 spec> <entry stmt>

| <sub1 impl> <other spec stmt>

| <sub1 impl> <spec stmt>

| <sub2 spec> <save stmt>

| <sub2 spec> <other spec stmt>

| <sub2 spec> <spec stmt>

| <sub2 spec> <parm stmt>

| <sub2 spec> <format stmt>

| <sub2 spec> <entry stmt>

<sub3 func> ::= <sub1 impl> <stmtfunc stmt>

| <sub1 impl> <data stmt>

| <sub2 spec> <stmtfunc stmt>

| <sub2 spec> <data stmt>

| <blok3 data> <entry stmt>

| <main3 func> <entry stmt>

| <sub3 func> <stmtfunc stmt>

| <sub3 func> <data stmt>

| <sub3 func> <format stmt>

| <sub3 func> <entry stmt>

<sub4 exec> ::= <sub1 impl> <exec stmt>

| <sub2 spec> <exec stmt>

| <sub3 func> <exec stmt>

| <main4 exec> <entry stmt>

| <sub4 exec> <exec stmt>

| <sub4 exec> <data stmt>

| <sub4 exec> <format stmt>

| <sub4 exec> <entry stmt>

<sub5 return> ::= <return stmt>

| <blok1 impl> <return stmt>

| <blok2 spec> <return stmt>

| <blok3 data> <return stmt>

| <main1 impl> <return stmt>

| <main2 spec> <return stmt>

| <main3 func> <return stmt>

```
                | <main4 exec> <return stmt>
                | <sub1 impl> <return stmt>
                | <sub2 spec> <return stmt>
                | <sub3 func> <return stmt>
                | <sub4 exec> <return stmt>
                | <sub5 return> <return stmt>
                | <sub5 return> <exec stmt>
                | <sub5 return> <data stmt>
                | <sub5 return> <format stmt>
                | <sub5 return> <entry stmt>
<spec stmt> ::= <dimen stmt>
              | <common stmt>
              | <equiv stmt>
              | <type stmt>
<other spec stmt> ::= <external stmt>
                    | <intrinsic stmt>
<exec stmt> ::= <assign stmt>
              | <goto stmt>
              | <arithif stmt>
              | <logif stmt>
              | <do stmt>
              | <continue stmt>
              | <stop stmt>
              | <pause stmt>
              | <read stmt>
              | <write stmt>
              | <print stmt>
```

| <rewind stmt>

| <backspace stmt>

| <endfile stmt>

| <open stmt>

| <close stmt>

| <inquire stmt>

| <call stmt>

BIBLIOGRAPHY

1. Aho, A. V., and Johnson, S. C., "LR Parsing", Computing Surveys, v. 6, n. 2, p. 99-124, June 1974.

2. American National Standard FORTRAN, ANS x3.9-1966, American National Standards Institute, 1966.

3. Digital Research, An Introduction to CP/M Features and Facilities, 1976.

4. Draft Proposed American National Standard FORTRAN, SIGPLAN Notices, v. 11, n. 3, March 1976.

5. Eubanks, G. E., A Microprocessor Implementation of Extended Basic, Masters Thesis, Naval Postgraduate School, December 1976.

6. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.

7. Intel Corporation, INSITE Library Programs 88-36, 8C-1, 8C-2, and 8C-4.

8. Knuth, D. E., "On the Translation of Languages from Left to Right", Information and Control, v. 8, p. 607-639, 1965.

9. Sammet, J. E., Programming Languages: History and Fundamentals, Prentice-Hall, 1969.

10. University of Toronto, Computer Systems Research Group Technical Report CSRG-2, "An Efficient LALR Parser Generator", by W. R. Lalonde, April 1971.

11. Yasaki, E. K., "The Emerging Microcomputer", DATAMATION, p. 81-86, December 1974.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	LT Lyle V. Rich, USN, Code 52Rs Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	LT Joan M. Russell, USN Naval Command Systems Support Activity Washington Navy Yard Washington, D.C. 20374	1

Thesis
R913
c.1

169964

Russell
An investigation of a FORTRAN grammar for use with a microprocessor based LALR (1) translator writing system.